Int J Softw Tools Technol Transfer (2015) 17:339–349
DOI 10.1007/s10009-014-0317-2

REGULAR PAPER

# Incremental test case generation using bounded model checking: an application to automatic rating

**Grzegorz Anielak · Grzegorz Jakacki · Sławomir Lasota**

Published online: 15 May 2014


**Abstract** In this paper we focus on the task of rating solutions to a programming exercise. State-of-the-art rating methods generally examine each solution against an exhaustive set of test cases, typically designed manually. Hence an issue of completeness arises. We propose the application of bounded model checking to the automatic generation of test cases. The experimental evaluation we have performed reveals a substantial increase in accuracy of ratings at a cost of a moderate increase in computation resources needed. Most importantly, application of model checking leads to the finding of errors in solutions that would previously have been classified as correct.

**Keywords** Bounded model checking · Test case generation · Program equivalence checking · Automatic rating

## 1 Introduction

Formal methods—in particular, model checking in its various forms [6]—find application in diverse domains. This paper, at a very general level, advocates the usefulness of formal methods in the process of automated rating. More concretely, it focuses on programming exercises, where programmers are asked to write a piece of code that satisfies some prescribed properties. Typical domains of application are programming contests (such as Google Code Jam, the ACM Collegiate Programming Contest, TopCoder Programming Contest, etc.), recruitment of programmers by software companies, or automatic grading of programming assignments in academic programming courses. Those domains have several characteristics that make them ideal targets for formal methods, and for bounded model checking in particular: known input constraints, a known correct solution, and a large body of relatively simple and short programs that are expected to share similar bugs.

An example programming exercise is shown in Sect. 5 below.

The rating of solutions is usually based exclusively on manually designed sets of test cases. On the one hand, the manual design of test cases often leads to highly incomplete coverage. On the other hand, automatic test case generation has been attracting an increasing amount of attention recently as a field of research [3,11]. From such a perspective, this paper may be considered an attempt to exploit automatic test case generation in order to improve the quality of automatic rating. It must be noted, however, that our objectives are quite different from usual in test case generation—a detailed comparison is deferred to the next section.

One particular feature of the rating scenarios discussed above is that a model solution of an exercise is usually available. (A rationale behind this is that a model solution appears very useful for validation of test cases.) This observation led us to the conclusion that the problem of rating may be seen as the problem of equivalence of programs: namely, the equivalence of the model solution and a submitted solution. Thus, at an abstract level, this paper is an attempt to answer the following question:

G. Anielak (✉) · S. Lasota
University of Warsaw, Warsaw, Poland
e-mail: grzegorz.anielak@gmail.com

S. Lasota
e-mail: sl@mimuw.edu.pl

G. Jakacki
Codility, London, UK
e-mail: jakacki@codility.com

Are formal methods applicable to the automatic derivation of test cases, on the basis of the nonequivalence of two programs?

Our answer is affirmative, as justified by the outcomes of the experimental evaluation to be explained below.

On the technical level, a contribution of the paper is the application of a model checker simultaneously for rating an examined program and for the generation of test cases for other programs to be examined in future. It is thus a combination of verification and test case generation: either a solution is rated as correct, or an error is found together with an illustrative counterexample, and a new test case is generated on the basis of the counterexample.

It is, however, worth emphasizing that the main contribution of the paper is not purely technical, as on the technical level we essentially build on the well-known method of bounded model checking. The main contribution of the paper is twofold: first, we identify a new application area for formal methods, namely automatic rating, which has remained relatively unexplored to date; second, we provide an experimental evaluation of our method, confirming a substantial increase in rating accuracy compared with rating based on manually designed test cases.

## 2 Overview of the approach

The formal method of our choice is bounded model checking (BMC) [4], a variant of model checking that assumes a predefined (but changeable) bound on length of computation.

Briefly, the approach we propose works as follows. A submitted solution is merged with the model solution; thus the program equivalence problem is reduced to the assertion violation problem in the merged program. Then a BMC engine is run on the merged program; if an assertion violation is found, the engine yields an example computation path resulting in the violation. The path is then inspected in order to obtain a computation path in the submitted solution that leads to the violation of some of the properties prescribed in the exercise.

In a case where a violation is found, the result is twofold. First, the submitted solution may be qualified as incorrect on the basis of the incorrect computation path produced. Second, the incorrect path is transformed into a test case, and thus improves coverage for subsequent solutions of the same exercise. We claim that the impact of the latter result is even greater than that of the former, as its usefulness is not restricted to one single solution but is spread across all the others. From such a perspective, our approach may be seen as a incremental variant of a model-based generation of test cases.

In order to answer affirmatively the question posed above, we performed an experimental evaluation of our BMC-based approach; the results are reported in Sect. 6, below. In the experiments we used the CBMC tool [5], and as input for the experiments we used the database of exercises and solutions obtained from an industrial tool for automatic rating developed by Codility.[1] All the solutions we used were written in the C language.

The experimental evaluation revealed a surprisingly high efficiency of model checking. On the one hand, bugs were found in a significant proportion of solutions that had previously been judged as being correct. These findings were made either directly, due to a violation discovered by CBMC, or indirectly, due to examination against test cases computed by former CBMC invocations. On the other hand, in the cases of a majority of solutions that had previously been found to be incorrect, the CBMC tool was also able to find a violation of correctness.

The impressive results of our experimental evaluation led to an immediate conclusion that model checking may be used jointly with the examination of test cases in the rating of solutions. Furthermore, counterexamples found by BMC are a valuable source of intricate new test cases, thus successfully complementing manual test case generation. Generally speaking, we provide further concrete evidence that model checking may be useful in finding subtle errors that are most often overlooked by programmers.

It is worth highlighting that the experimental evaluation was performed on real-life solutions submitted by individuals in real-life conditions (pre-hire screening), which makes it a very valuable contribution. Instead of exploiting some contrived examples to argue that the method works, we examine pieces of code that come straight from the field, written by real people under real, not artificial, test conditions.

The remaining part of the paper is organised as follows. In Sect. 3 we present details of our approach. For illustration we use an example exercise from the Codility database and carry out an example falsification to describe all the steps of the procedure. We also briefly present the workings of CBMC to familiarize the reader with the idea of bounded model checking as applied to the C language.

In Sect. 4 we summarize the principal advantages of our method in the context of automatic rating of programming exercises. We also discuss some potential limitations of our approach.

In Sect. 5 we investigate a genuine, submitted solution that had already been classified as correct, but which was falsified by our method. The aim of this section is not merely to instantiate and illustrate the procedure with a concrete example, but rather to point out the subtlety of errors that may be overlooked by human test case designers.

[1] Codility.com is a commercial service for screening the programming skills of individuals through the automatic assessment of solutions to short programming exercises.

Finally, in Sect. 6 we describe the experiments performed, and report in detail the results of our experimental evaluation.

This paper is based on the Master's thesis of the first author [2].

*Related research* As far as we are aware, very little is known about the usefulness of formal methods in automatic rating, such as programming assignment assessment, or programming contest rating. One proposal [17] suggests the automation of assessment of students' formal specification coursework. The approach is based on some ad-hoc metrics, while our approach is based on a rigorous application of a model checker.

There is a widely established consensus that formal methods and testing may be seen as complementary methods in the development of high-quality software. Furthermore, the possibility of application of model checkers for *automated* program testing is receiving increasing attention in the literature; see, for instance, [1,3,9,11–14]. The strongest point of model checking is its capacity for the fully automated generation of test suites (see, for instance, [3]); on the other hand, some drawbacks resulting from the use of model checkers in test case generation are discussed in [9], originating in the fact that software model checking is not in a fully mature state yet. Below, we consider briefly some chosen examples of such applications, focusing on those involving the CBMC tool, and then relate their methods and objectives to ours. For a more comprehensive treatment of the current state of the art, we refer the reader e.g. to [11].

As reported in [1], the CBMC tool has been used as a test generator for the coverage analysis (cf. [3]) of software, yielding a substantial reduction in costs of the testing phase in an industrial setting. A similar increase of coverage is described in [12]. A slightly different approach, where the model checker is used directly for unit testing, is presented in [13]. Finally, the authors of [14] provide a comparative study of applying different model checkers to component testing, from the viewpoint of a real-world industrial project.

Compared with the aforementioned area of research, our objectives are different as we are mostly interested in the *incremental* scenario of test case generation. In our approach, the aim of application of a model checker is twofold: first, to find a bug in a currently examined solution, and second, when a bug is found, to automatically derive a test case for other solutions, including those examined in future. We are thus dealing not with a single solution program but with a continuously augmenting collection of solutions, and therefore, instead of applying standard coverage metrics for a single program, it would only be reasonable to measure coverage of the space of potential erroneous solutions.

Our approach assumes the existence of a model solution, and essentially amounts to checking the *equivalence* of a given solution and a model one. The idea of reduction of a verification problem to equivalence checking appears naturally in many contexts and is by no means new. Even the CBMC tool itself has been used successfully for equivalence checking of programs, and equivalence of a program and a circuit; see, for instance [15,16].

The assumption about the existence of a model solution resembles the scenario of *model-based testing* [7,18]. In the latter approach, a model of a system is built in a way that reflects the principal functionalities of the system tested, and then the model is used to derive test suites in an automated way. Diverse formalisms have been exploited for model description, including finite-state automata, certain UML diagrams, Markov chains, logical formalisms like Z, etc. Depending on the choice of formalism, different techniques have been applied for test case generation; for an overview see [18] and the literature mentioned therein. Among other techniques, model checking tools have been also used in the process of test case generation; see e.g. [10]. In a typical scenario, a model checker is provided with a model of the system under test and a property of the system to be tested. Then the test suites are produced from the results of model checking; for instance, from counterexample paths in the model when the tested property is violated by the model.

In our approach, a model checker is used not only in order to validate an examined solution, but also in order to generate new test cases for other solutions which resembles the model checking-based test generation as described above. We note, however, an important technical difference. In model-based testing, test cases are typically derived from a model and a property to be tested. On the other hand, in our approach, new test cases are derived exclusively from instances of *inequivalence* of an examined solution and a model solution. In particular, no additional effort is required to build a specification of an expected property.

## 3 Methodology

Consider the following simple programming exercise.

> *Write a function*
>
> `int equi (int *A, int n)`
>
> *that, for a given nonempty zero-indexed array of integers* `A`*, returns the equilibrium point, i.e. the minimal position k that verifies the property* $\Sigma_{i<k}\,\mathtt{A}[i] = \Sigma_{i>k}\,\mathtt{A}[i]$*. If* `A` *has no equilibrium point, the function should return* $-1$*.*

For instance, if the array `A` contains the following numbers:

$$-7 \quad 1 \quad 4 \quad 2 \quad -3 \quad 2 \quad 1 \quad -2 \quad 0,$$

the result of function `equi` should be 3.

We will now walk through an example application of our method and present the detailed workings of our approach. Before we can falsify any solutions, we need to acquire two C functions: `check_input` and `model_equi`. These are two pieces of code prepared once for the exercise and then reused without changes for every submitted solution.

The first one, `check_input`, is a function that checks whether the given input parameters meet the constraints stated in the exercise description. This is needed to ensure that we only find counterexamples within the exercise domain. We use a `CPROVER_assume` statement that instructs the CBMC tool to restrict computation paths only to those satisfying a given condition. For example, the `equi` exercise works with nonempty arrays, which would be reflected in the `check_input` function as presented below.

```
void check_input(int *A, int n) {
  CPROVER_assume(n > 0);
}
```

The second function, `model_equi`, provides a model solution for the exercise, i.e. the solution intended to be correct. Besides being correct, the model should also be efficient (see Sect. 4.1). Such a solution has to be devised manually by a skilled programmer. However, our experience suggests that, in the domain of programming assignments and contests, a model solution is usually available upfront, as it proves to be very useful for test-case-based rating alone. We assume that the model solution is valid (see Sect. 4.3).

Below we present a model solution for the `equi` exercise.

```
int model_equi(int *A, int n) {
  int i;
  long long t = 0;

  for(i = 0; i < n; i++)
    t += A[i];

  for(i = 0; i < n; i++) {
    if (t == A[i])
      return i;
    t -= 2LL * A[i];
  }
  return -1;
}
```

Having both `check_input` and `model_equi` functions, we are now ready to falsify user solutions. In order to do so, we evaluate the equivalence of the user and the model implementation. Consider the particularly simple solution presented below. The program claims (obviously wrongly) that there is always an equilibrium point at position 0.

```
int user_equi(int *A, int n) {
  return 0;
}
```

We will find the program incorrect using our method. The procedure operates along the following lines:

1. A single C program is constructed by putting the user and model solutions together. The program has an assertion, so that its violation corresponds to the non-equivalence of the `model_equi` and `user_equi` functions.

```
void check_input(int *A, int n){...}
int model_equi(int *A, int n){...}
int user_equi(int *A, int n){...}

int main() {
  // Non-deterministic variables.
  // Any values will be considered
  // for those.
  int n;
  int *A = malloc(n * sizeof(int));

 //Limiting paths with the exercise
 // constraints.
 check_input(A, n);

  // Asserting equivalence.
  assert(model_equi(A, n) ==
           user_equi(A, n));

  return 0;
}
```

2. The CBMC tool is run against the merged program to look for a computation path that violates the assertion. We bound the length of the paths considered by specifying to CBMC the number of times loops should be unwound (see Sect. 3.2, below). We then iteratively deepen our search (i.e. consider longer and longer computation paths) until an assertion violation is found, or we reach the time-limit intended for the task. For example, note that the `user_equi` function presented above returns the correct answer for all single-element sequences. This means that non-equivalence would only be discovered once we considered computation paths that are long enough to handle sequences of two elements.
3. If a path that violates the assertion is found, it is transformed into a counterexample. Basically, we have to filter the values to which uninitialized variables (like `n` and `A` in this case) were set in this particular path. In our example case, we discover that the violating path is setting `n = 2` and `A = [0, 1]`.

4. The counterexample, if found, is verified. By this we mean separately compiling both solutions and running them against the counterexample. We check whether condition

```
model_equi(< counterexample >)
                ≠
user_equi(< counterexample >)
```

holds. We run the programs and see that for A = [0, 1] and n = 2, `user_equi(A, n)` is 0, while `model_equi(A, n)` is −1. This is a confirmation that the counterexample is valid.
5. Finally, provided the counterexample passed the verification above, a test case is constructed out of it. All past and future solutions to the exercise will be run against it.

### 3.1 Possible outcomes

To summarize, the falsification may yield four different results, as presented below. Discussion of the distribution of the results observed in the course of the experiments is deferred to Sect. 6.

*No counterexample* The tool has hit the time limit and no counterexample has been found. Thus, either the solution is correct or the counterexamples are too complex to be found within the available time.

*Counterexample* The tool has reported a valid counterexample, i.e. it has found input data within the exercise constraints for which the solution gives a wrong answer (different than the model).

*False counterexample* The tool has reported an invalid counterexample, i.e. input data for which the solution nevertheless returns a correct answer (as checked in step 4 above). Theoretically such an inconsistency should not happen, but it may arise in practice due to technical reasons. Namely, we falsify solutions using CBMC while later we examine executables produced by the gcc compiler, and the semantics used by CBMC and gcc happen to differ slightly. One particular example is usage of language constructs with undefined results, e.g. uninitialized variables, when a solution may work only by accident, and when it does we cannot detect it. Counterexample falsehood is usually dependent on the setup: architecture, versions of CBMC and gcc, etc.

*Error* The falsification has been interrupted by some error before any counterexample has been found. Some possible errors include: hitting a memory limit, nonstandard syntax of the source code, or bugs in the CBMC tool.

### 3.2 Bounded model checking with CBMC

As we have seen, we reduce the equivalence problem to finding an assertion violation in the single program. At that point, we hand over most of the work to the BMC tool of our choice—CBMC. The tool uses bounded model checking to verify properties (like assertion violation) of C programs.

In short, the tool reduces the assertion violation problem to the problem of boolean formula satisfiability (SAT). An input program is translated into a boolean formula, so that satisfiability of the formula corresponds to a violation of the assertion in question. Getting it to work with all the features of the C language is quite a lot of work, and we are thankful that we could rely on the external tool in that respect. Below we only sketch this transformation, to give the reader a general idea. More details can be found in [5] and [15].

Consider a C program with an assertion. We want to find a computation path that violates the assertion. First, the length of the considered computation paths is bound (hence *bounded* model checking) by the procedure called *loop unwinding* (i.e. replacing the loop by the code inside the loop duplicated a specified number of times). This makes the set of possible computation paths finite by limiting them to the ones that would execute the loop at most this number of times. The same happens to the other sources of looped computation, such as recursion or `goto` statements. Here is an example of unwinding the loop

```
while (e) {
  ... body code ...
}
```

three times:

```
if (e) {
  ... body code ...
  if (e) {
    ... body code ...
    if (e) {
      ... body code ...
      CPROVER_assume(!e);
    }
  }
}
```

We use constraint `CPROVER_assume(!e)` to discard the paths that would execute the loop further. It is also common to use assertion `assert(!e)` instead. The latter is especially useful for verification, i.e. proving the program correct. Longer computation paths could spoil the proof, so it has to be asserted that there are none of them. This is not the case for us: we run falsification, i.e. we do not provide proof of correctness, but search extensively for counterexamples. We focus on the current bound and accept the fact that there might exist longer computation paths (that we might

or might not look over later). The CBMC tool provides a command-line switch to control which behavior is used (the switch `–no-unwinding-assertions`).

The next step is to rename the variables so that each variable is assigned only once (like a constant), which makes a formula easier to build. This is achieved by creating a copy of a variable every time it is assigned. See the example of transformation below.

```
x = x + 1;        ⟶    x1 = x0 + 1;
y = x + y;              y1 = x1 + y0;
```

Once we have loops unwound and variables renamed, the program is encoded into a boolean formula. Define two functions: $C(p, g)$ (for constraints) and $P(p, g)$ (for properties). Both take a program $p$ and a guard $g$ (a condition that is true at the beginning of $p$), and map it to a boolean formula. Both are defined by induction over the syntax of program $p$. Below we provide the definition of functions $C$ and $P$ for some chosen common programming constructs.

*Empty program*

$$C(\texttt{''}, g) = true$$

$$P(\texttt{''}, g) = true$$

*Conditional*

$$C(\texttt{'if (c) \{ I \}'}, g) = C(I, g \wedge c)$$

$$P(\texttt{'if (c) \{ I \}'}, g) = P(I, g \wedge c)$$

*Sequential composition*

$$C(\texttt{'I; J'}, g) = C(I, g) \wedge C(J, g)$$

$$P(\texttt{'I; J'}, g) = P(I, g) \wedge P(J, g)$$

*Assignment*

$$C(\texttt{'x}_i \texttt{ = e'}, g) = (g \wedge x_i = e) \vee (\neg g \wedge x_i = x_{i-1})$$

$$P(\texttt{'x}_i \texttt{ = e'}, g) = (g \wedge x_i = e) \vee (\neg g \wedge x_i = x_{i-1})$$

*Assertion*

$$C(\texttt{'assert(c)'}, g) = true$$

$$P(\texttt{'assert(c)'}, g) = g \rightarrow c$$

*Constraint*

$$C(\texttt{'CPROVER\_assume(c)'}, g) = g \rightarrow c$$

$$P(\texttt{'CPROVER\_assume(c)'}, g) = true$$

Finally, consider the formula $C(p, true) \wedge \neg P(p, true)$, where $p$ is the whole program. If the formula is satisfiable, we have found a computation path that meets all the constraints but violates one of the assertions (properties). We evaluate the satisfiability by passing the formula to the SAT solver. The CBMC tool supports a variety of SAT solvers; for our experiments we have used MiniSAT [8].

If the formula is not satisfiable, either the assertion cannot be violated or it can only be violated by computation paths longer than the ones we considered (i.e. the loops have not been unwound enough times). We can try unwinding the loops further and repeat the whole process. In our application, we do this until we exceed some predefined time-limit.

Apart from the simple programming constructs mentioned here, the CBMC tool supports other, more complicated features of the C language, including arrays and pointers. More on the subject of language coverage is deferred to Sect. 4.4.

## 4 Limitations

We believe that our approach is especially suitable for automatic rating in the domain of programming contests, programming assignments in university classes and pre-hire screening of programmers. These domains reveal a specific set of features that very well match the requirements of our approach. Among those requirements are:

- Relatively simple and short programs that make exponential methods of bounded model checking feasible;
- Known input constraints;
- Availability of a correct solution;
- An augmenting set of programs that solve the same problem; the programs are expected to have similar bugs, which means that their falsification is not completely independent (counterexamples might be successfully reused, as was the case in our application).

On the other hand, our approach is susceptible to various potential limitations. Below we discuss some major sources of limitations we observe.

### 4.1 Inefficient solutions

Due to the nature of bounded model checking, the more efficient a solution is, the more practical the efforts to falsify it will be. Inefficient implementations tend to have their computation paths grow quickly with the size of the input, which automatically makes finding a path constituting a counterexample more difficult. In particular, we are helpless in the face of hanging solutions—their computation paths do not even reach the assertion, let alone violate it. Extremely inefficient solutions must be filtered by other means (e.g. benchmarking on large test cases) when performing automating rating.

### 4.2 Model availability

As mentioned already, our method assumes access to a model solution. We imagine this requirement to be quite restrictive in other industrial applications of formal verification. However, it is not so in the domain of programming contests, pre-hire screening and university assignments. For those, a correct solution is usually available upfront, as it is common to create one as part of an exercise design. There are at least two reasons for that. First, the model works as a proof of concept—it convinces the exercise designer that his intended solution really works. Second, the model proves to be useful in designing test cases for the exercise. Apart from double-checking the small test cases, it may be the only practical way to get the expected correct answers for larger inputs.

As an example, we did not have to come up with any model solution during our experimentation. Codility has them for every programming exercise that it provides.

### 4.3 Invalid models

Correctness of the model is our crucial assumption. Having detected non-equivalence, we know that either the model or the user solution is incorrect. We always assume the latter; thus a faulty model compromises our approach. Unusual results arising from this situation might be noticed (e.g. all submitted solutions are falsified), but detection is by no means automatic.

As mentioned above, we are reusing existing model solutions that have already been working in the field for a substantial amount of time. That helps a lot in terms of confidence in the models' quality. The chance of their containing mistakes is low, as mistakes have largely been eliminated over the years (e.g. due to users' complaints).

### 4.4 Language coverage

Since we try to falsify arbitrary programs written in C, a natural question arises: how good are our tools at understanding the language?

The coverage of language features in our approach depends mostly on the coverage of the BMC engine used. We rate the coverage of the CBMC tool very highly, especially for the features that are part of the ANSI-C standard. Supported features include functions (including recursive functions, which are unwound similarly to loops), arrays (including dynamically-sized and multi-dimensional arrays), pointers, pointer arithmetic, dynamic memory allocation and structures.

Additionally, we perform counterexample validation (see step 4 in Sect. 3): i.e. we confirm that the counterexample works in the usual runtime environment. This eliminates the possibility that counterexample comes solely from a C semantics flaw in the BMC engine. We also believe that our results prove that the tools have properly understood most of the programs.

The limitation of our approach, as based on CBMC, is that it only works with C/C++[2] programs. In principle, the analogous approach should work equally well for other languages, but it would require the adaptation of another BMC engine or translation from the language of choice to C.

## 5 Example errors

Consider the following solution to the `equi` exercise, which was actually submitted to the Codility system in the past, and was classified as correct.

```
int equi(int *A, int n) {
    int i;
    long long t = 0;

    for(i = 0; i < n; i++)
        t += A[i];

    for(i = 0; i < n; i++) {
        if (t == A[i])
            return i;
        t -= 2 * A[i];
    }
    return -1;
}
```

The solution is surprisingly similar to the solution that we presented as model in Sect. 3. However, the solution contains a subtle error, in the instruction

```
t -= 2 * A[i];
```

that updates the variable `t`. Namely, the right-hand side of the assignment is evaluated within type `int`, and may cause an arithmetic overflow. Indeed, this actually happens, for instance when the input array contains the following data (where `INT_MAX` is the maximum value `int` can hold):

```
INT_MAX 0 INT_MAX.
```

An appropriate counterexample was automatically found in a few seconds, as shown in the invocation below:

```
$ bmc-check submitted.c model.c
counterexample:[[1384378608,0,
  1384378608]]
counterexample confirmed: got -1,
  but equilibrium point exists,
  e.g. on position 1
overall time: 3.154 s.
```

[2] Some support of C++ is claimed by CBMC authors. We have not tried that, as we have focused on C programs only.

The test case deduced from the above counterexample has since been added to the Codility database.

Another interesting example of a mistake often made by programmers is use of a floating-point data type instead of integers. Consider the following program (derived and simplified from actual solutions from the Codility database that were classified as correct):

```
int equi(int *A, int n) {
    int i;
    float s = 0, t = 0;

    for(i = 0; i < n; i++)
        t += A[i];

    for(i = 0; i < n; i++) {
        if (t - s - A[i] == s)
            return i;
        s += A[i];
    }
    return -1;
}
```

The solution is incorrect due to its use of the type `float`, the precision of which is insufficient for this purpose. The error can be revealed by a carefully designed counterexample involving both very big and very small numbers; nevertheless, a number of solutions similar to that shown above passed the manually-designed test cases. By contrast, our method is very sensitive to this kind of error, and easily produces a valid counterexample.

```
$ bmc-check submitted.c model.c
counterexample: [[-2147483648, 16]]
counterexample confirmed: got 0,
  but it is not equilibrium point
overall time: 2.589 s.
```

The counterexample is based on the surprising observation that when computations are performed in type `float`, one gets:

```
-2147483648 + 16  = -2147483648.
```

## 6 Experimental evaluation

We carried out rigorous tests to measure the effectiveness and usefulness of our method. In this section we present the results of our experiments.

### 6.1 Methodology

As input for our method we used programs from the Codility database, i.e. solutions to exercises submitted by users over several years. We included all solutions written in C for the exercises that we already support. We excluded solutions that are trivially invalid, i.e. that do not compile or cannot solve any of the Codility test cases. This leaves a subset containing 10,312 solutions solutions from 14 exercises.

In our experiments, we compare rating results from the rating system used originally in the Codility tool with rating results from the BMC-based method proposed by us.

The experiments were conducted on an Intel Core i7 (2.2 GHz) machine. The tools were run on a single core with a time limit of 180 s and a memory limit of 2 GB.

As a BMC engine we chose the CBMC tool [5]. The results we obtained were compared to the original Codility rating system (manually designed test cases). We conducted different kinds of comparisons, which led to our two main experiments:

I. We tried to falsify solutions that were recognized as correct at the time of submission. This enabled us to measure whether Codility could benefit from application of BMC.
II. We tried to falsify solutions that were already known to be incorrect. This enabled us to estimate how tight our method is, i.e. what proportion of bugs it misses.

We treat our method as a test case generation technique. This means that whenever CBMC produces a counterexample for some particular solution, we also examine other solutions against the same counterexample. The justification for this is that solutions written by distinct authors often share similar bugs, and an individual counterexample can reveal faults within many similar solutions. We find this technique beneficial, as can be seen in the results presented below. If not stated otherwise, solutions compromised by such foreign counterexamples are treated as having been falsified by our method, even if we could not falsify them directly using BMC.

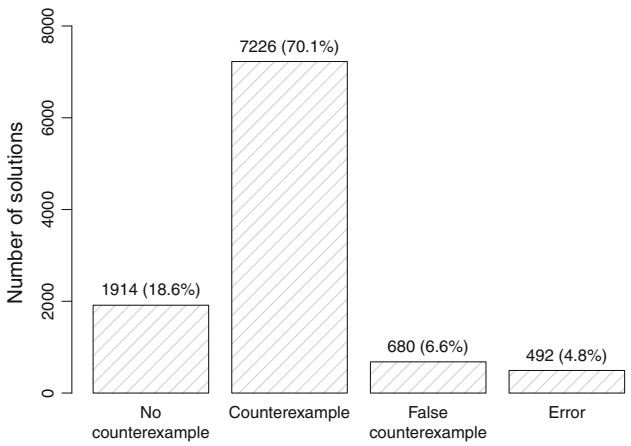


**Fig. 1** Distribution of results among all the falsifications we performed

Note that the sets of solutions used in both experiments may not be disjoint. The first experiment uses solutions that were considered to be correct at the time of submission (but not necessarily now). If such a solution has later been recognized as faulty (for example, by new manual test cases), it will be used in both experiments. The rationale for Experiment I is to answer the question of how an automated rating system could benefit from BMC if it were integrated from the beginning of the system's existence.

As noted in Sect. 3.1, the falsification may yield four different results: no counterexample, counterexample, false counterexample, or error. For distribution of these results see Fig. 1.

### 6.2 Experiment I

In this experiment we tested solutions that the original rating system considered to be fully correct at the time of submission. We tested 1,518 such solutions. We were able to falsify 114 out of the total of 1,518 solutions, which constitutes about 7.5 % (see Fig. 2).

We would like to emphasize the relevance of these results. The percentage of falsified solutions may not look very impressive, but in fact it really is highly significant! Note that all the solutions investigated in the experiment had already been rigorously tested by other methods. Furthermore, the quality of the test case, being a part of a professional rating system, was very high. Thus, most of the solutions are probably really correct and cannot be falsified by any means.

On the other hand, differentiating between correct and incorrect solutions is crucial for the accuracy of automatic rating. When a solution is recognized as being correct, its author receives the highest possible score: something we would definitely like to avoid if the solution is in fact faulty.

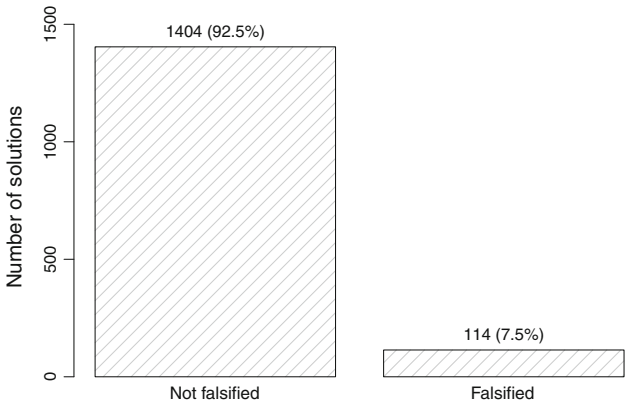


**Fig. 2** Experiment I—falsifying 'correct' solutions. 'Not falsified' represents solutions that, to the best of our knowledge, are correct—they were falsified neither by the original rating system nor by our tool based on BMC. 'Falsified' represents solutions that were falsified by us, even though they were rated as fully correct at the time of submission

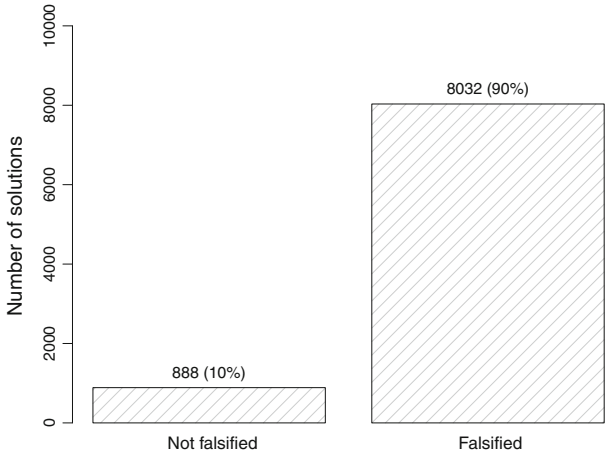


**Fig. 3** Experiment II—falsifying incorrect solutions. 'Not falsified' represents solutions that we were not able to falsify, even though they were known to be incorrect. 'Falsified' represents incorrect solutions that were properly falsified by our method

Nevertheless, despite the high quality of test cases, some faulty solutions still pass through. We thus conclude that bounded model checking can significantly help in closing this gap.

### 6.3 Experiment II

In this experiment we tested solutions that were already known to be incorrect, e.g. that were falsified by test cases. We tested 8,920 such incorrect solutions. Our method was able to reveal the faultiness of 8,920 solutions, which constitutes about 90 % (see Fig. 3).

### 6.4 Counterexample propagation

As we have explained, we use BMC not only for falsification but also for the generation of test cases, which then can be used to examine other solutions. We have analyzed the effectiveness of this technique by relating the number of solutions that we would be able to falsify only with direct application of BMC to the total number of solutions falsified by us.

From 8,032 solutions that we successfully falsified, 806 were falsified thanks to counterexample propagation; i.e. they were falsified using counterexamples obtained from model checking other solutions (see Fig. 4). This constitutes around 10 %. The result shows that, although direct application of BMC yields a majority of falsifications, running all the counterexamples against all the solutions can still make a significant positive difference.

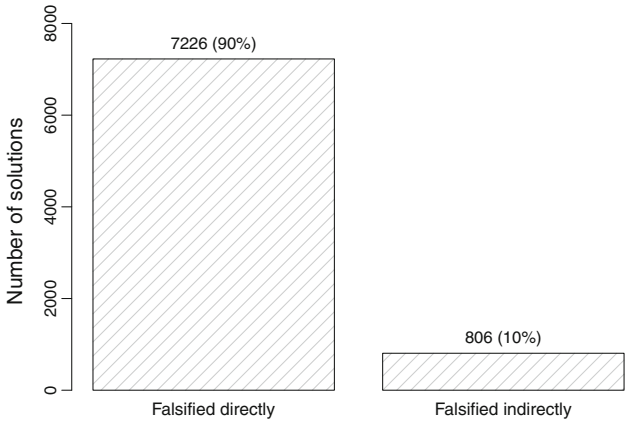


**Fig. 4** Counterexample propagation. 'Falsified directly' represents solutions that were falsified directly using BMC. 'Falsified indirectly' represents those solutions for which CBMC did not produce any counterexample within the predefined time limit, but which fail on one of the counterexamples obtained from other solutions

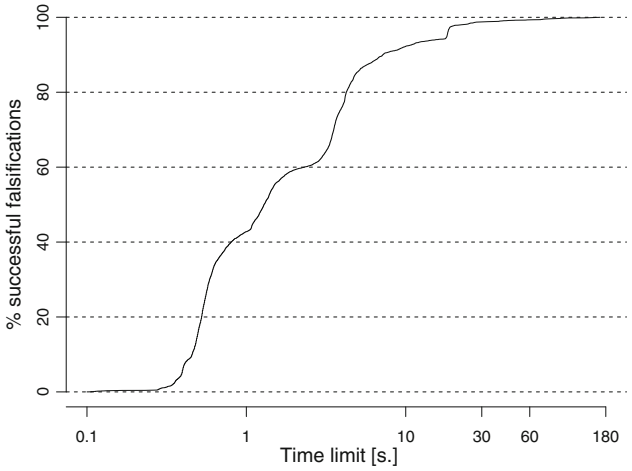


**Fig. 5** Percentage of successful falsifications that would be possible depending on the time limit setting. For a given time $t$, the percentage is calculated as the proportion of the number of falsifications that succeeded before time $t$ and the number of all successful falsifications. The time limit axis is logarithmic

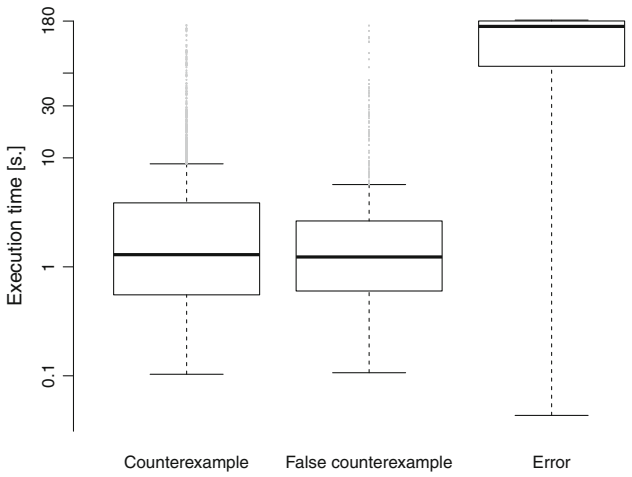


**Fig. 6** Distribution of the falsification execution time depending on its result. For the 'no counterexample' result the execution time is always equal to the time limit, i.e. 180 s

## 6.5 Time cost

One of the main parameters of our tool is a time limit—a maximum time that may be used for the falsification of a single solution. If a counterexample cannot be found within a given time, we simply interrupt falsification and classify the solution as 'not falsified'. We gathered and analyzed the execution times of successful falsifications. Such analysis gives us insight into the time constraints of our method, as well as hints for the proper setting of time limits in future. For all of our experiments we set a time limit of 180 s.

For a given time $t$, we calculate what would happen if the time limit were set to $t$; that is, we calculate the percentage of successful falsifications that took less time than $t$. Such a percentage represents the usefulness of setting the time limit to at least $t$.

We have discovered that the vast majority of successful falsifications (more than 98 %) succeed in the first 30 s (see Fig. 5). Moreover, this is also true of the falsifications that we care most about, namely falsifications from Experiment I—96 % of which would also be found within a time limit of 30 s.

For distribution of the execution time depending on the falsification result, see Fig. 6. The result 'no counterexample' is not included as it always, by definition, occurs when the time limit is reached.

The data show that our method is suitable also for time-sensitive applications, such as falsifying a solution immediately after its submission, while the candidate is waiting for his/her score. With the time limit set to 30 s (or even less), one could still benefit greatly from bounded model checking. On the other hand, there are still some counterexamples that require a much longer time to be found. Therefore, a higher time limit seems a reasonable choice for applications where time is not a prime concern, such as test case generation on the basis of previously submitted solutions.

## 6.6 Historical data

According to our experimental data, an automated rating system would benefit from the continuous application of formal methods throughout its whole life cycle. For a given moment in time, the added value provided by application of our method is estimated by the number of solutions that our method would falsify compared to the original system. The results, shown in Fig. 7, indicate that our method constantly outperforms the original system, despite the fact that manual test case coverage also increases with time. Actually, Fig. 7

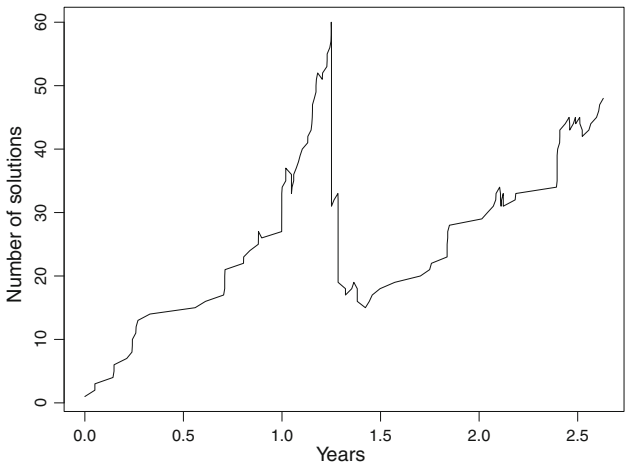


**Fig. 7** The added value of our method, presented as a function of time. For a given moment in time, the chart shows the number of solutions that would be falsified by our method, but which were at that moment considered correct

underestimates the impact of our method, as the original system has not been completely isolated from our research. For instance, the big drop seen in the graph corresponds to the introduction of a new set of test cases based on some of the counterexamples we found previously. Rapid upturns seen in the figures are a result of submissions of new solutions that led to counterexamples which also falsify many previous solutions. Once again, the results show the positive effect of counterexample propagation.

## 7 Conclusions

This paper reports on a success story concerning the application of formal methods in the field of automatic rating. We argue that the assessment of programming exercises can be substantially improved through the application of bounded model checking (BMC). Importantly, the improvement has been confirmed experimentally, using real-life data obtained from a commercial code assessment service (Codility). Furthermore, a counterexample produced by BMC for one solution becomes a valuable source of test cases, useful for falsifying other solutions (counterexample propagation). Thus we also demonstrate that BMC may be used naturally for the incremental generation of test cases.

**Acknowledgments** The last author acknowledges a partial support of the National Science Centre Grant 2013/09/B/ST6/01575.



## References

1. Angeletti, D., Giunchiglia, E., Narizzano, M., Puddu, A., Sabina, S.: Using bounded model checking for coverage analysis of safety-critical software in an industrial setting. J. Autom. Reason. **45**(4), 397–414 (2010)
2. Anielak, G.: Sprawdzanie równoważności programów przy użyciu ograniczonej weryfikacji modelowej. Master's thesis, University of Warsaw. (In Polish) (2012)
3. Ball, T.: A theory of predicate-complete test coverage and generation. In: FMCO, pp. 1–22 (2004)
4. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Adv. Comput. **58**, 118–149 (2003)
5. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: K. Jensen, A. Podelski (eds.) Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004), Lecture Notes in Computer Science, vol. 2988, pp. 168–176. Springer, Berlin (2004)
6. Clarke Jr, E.M., Grumberg, O., Peled, D.A.: Model checking. MIT Press, Cambridge (1999)
7. Dias-Neto, A.C., Travassos, G.H.: Model-based testing approaches selection for software projects. Inf. Softw. Technol. **51**(11), 1487–1504 (2009)
8. Eén, N., Sörensson, N.: An extensible sat-solver. In: E. Giunchiglia, A. Tacchella (eds.) SAT, Lecture Notes in Computer Science, vol. 2919, pp. 502–518. Springer, Berlin (2003)
9. Fraser, G., Wotawa, F., Ammann, P.: Issues in using model checkers for test case generation. J. Syst. Softw. **82**(9), 1403–1418 (2009)
10. Fraser, G., Wotawa, F., Ammann, P.E.: Testing with model checkers: a survey. Softw. Test. Verif. Reliab. **19**(3), 215–261 (2009)
11. Hierons, R.M., Bogdanov, K., Bowen, J.P., Cleaveland, R., Derrick, J., Dick, J., Gheorghe, M., Harman, M., Kapoor, K., Krause, P., Lüttgen, G., Simons, A.J.H., Vilkomir, S., Woodward, M.R., Zedan, H.: Using formal specifications to support testing. ACM Comput. Surv. 41(2), 9:1–9:76 (2009)
12. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: Query-driven program testing. In: N. Jones, M. Mller-Olm (eds.) Verification, Model Checking, and Abstract Interpretation, Lecture Notes in Computer Science, vol. 5403, pp. 151–166 (2009)
13. Kim, M., Kim, Y., Kim, H.: Unit testing of flash memory device driver through a sat-based model checker. In: ASE 2008. 23rd IEEE/ACM International Conference on Automated Software Engineering (2008)
14. Kim, M., Kim, Y., Kim, H.: A comparative study of software model checkers as unit testing tools: an industrial case study. IEEE Trans. Softw. Eng. **37**(2), 146–160 (2011)
15. Kroening, D., Clarke, E., Yorav, K.: Behavioral consistency of C and Verilog programs using bounded model checking. In: Proceedings of DAC 2003, pp. 368–371. ACM Press, New York (2003)
16. Lee, D.A., Yoo, J., Lee, J.S.: Equivalence checking between function block diagrams and C programs using HW-CBMC. In: Proceedings of the 30th International Conference on Computer Safety, Reliability, and Security, SAFECOMP'11, pp. 397–408. Springer, Berlin (2011)
17. Shukur, Z., Burke, E., Foxley, E.: The automatic assessment of formal specification coursework. J. Comput. High. Educ. **11**(1), 86–119 (1999)
18. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan-Kaufmann, Burlington (2007)